CSC263 Assignment #3

You are allowed to work in a group of at most 3 members, and make one submission as a group. You should clearly write names of all group members in the first page of your submission. Everyone in the same group will get the same marks.
Please read and understand the policy on Academic Honesty on the course website. Then, to protect yourself, list on the front of your submission every source of information you used to complete this homework (other than your own lecture and tutorial notes). For example, indicate clearly the name of every student with whom you had discussions, the title and sections of every textbook you consulted (including the course textbook), the source of every web document you used (including documents from the course webpage), etc.
For each question, please write up detailed answers carefully. Make sure that you use notation and terminology correctly, and that you explain and justify what you are doing. Marks will be deducted for incorrect or ambiguous use of notation and terminology, and for making incorrect, unjustified, ambiguous, or vague claims in your solutions.

From this assignment on, most questions on writing algorithms will not explicitly require pseudocode. You may either describe in english, or write pseudocode, or give both (for example, write pseudocode for sub-procedures). When you describe your algorithm, write concisely and precisely in a few lines; you may also itemize the algorithm in several steps.
We will also give some simple exercise questions, which are similar to the ones in the midterm.

1. (*5 points*) Linear probing in open-addressing hash table.

Consider an open-addressing hash table that uses the hash function

$h(k, i) = (h'(k) + i) \mod 13$,
$h'(k) = k \mod 13$.

Fill in the slots below by inserting the following sequence of keys to the hash table defined above:

- Already inserted: 10, 8, 130, 13
- To insert: 27, 9, 14, 23, 21

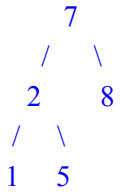
slot	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
key	130	13							8		10		

slot	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
key	130	13	27	14					8	9	10	23	21

2. (*5 points*) AVL tree insertion and deletion.

(a) Draw the AVL tree after inserting the following sequence of keys into an empty tree:
5, 8, 7, 1, 2
(b) Draw the AVL tree after deleting 8 from the AVL tree obtained in (a).

(a)



(b)

3. (*10 points*) Amortized analysis of deletion in dynamic arrays using the potential method.
Suppose that instead of shrinking a table by halving its size when its load factor drops below 1/4, we shrink it by multiplying its size by 2/3 when its load factor drops below 1/3. Using the potential function

$$\Phi(T) = |\, 2\, T.num - T.size\,|,$$

show that the amortized cost of deletion that uses this strategy is bounded above by a constant.

Suppose before a deletion operation, we have $n = T.num$, $m = T.size$, and the load factor $\alpha = n/m$. Let T' be the structure after the deletion, and α' be its load factor. Let c be the actual cost of the deletion, and $\hat{c}$ be the amortized cost. Then $\hat{c} = \Phi(T') - \Phi(T) + c$.

We consider different cases of α.

- If $\alpha > ½$, then
  $\Phi(T) = 2n - m$, $\Phi(T') = 2(n-1) - m$, $c = 1$
  $\hat{c} = -1$
- If $⅓ < \alpha \le ½$, then
  $\Phi(T) = m - 2n$, $\Phi(T') = m - 2(n-1)$, $c = 1$
  $\hat{c} = 3$
- If $\alpha = ⅓$, then $n = ⅓\, m$, and
  $\Phi(T) = m - 2n$, $\Phi(T') = ⅔\, m - 2(n-1)$, $c = n$
  $\hat{c} = 2$

Therefore, in all cases the amortized cost is $O(1)$.

4. (1*5 points*) Bit-reversed binary counter.
Consider a ***bit-reversal permutation*** on an input array $B[0..n-1]$ whose length is $n = 2^k$ for some nonnegative integer k. This permutation swaps elements whose indices have binary representations that are the reverse of each other.

We can express each index a as a k-bit sequence $(a_{k-1}, a_{k-2}, \ldots, a_0)$, where

$a = \sum_{i=0,\ldots,k-1} a_i 2^i$.

We define

$\text{rev}_k(a_{k-1}, a_{k-2}, \ldots, a_0) = (a_0, a_1, \ldots, a_{k-1})$.

Thus we have

$\text{rev}_k(a) = \sum_{i=0,\ldots,k-1} a_{k-i-1} 2^i$.

Note that

$\text{rev}_k(\text{rev}_k(a)) = a$.

For example, if $n = 16$ (or, equivalently, $k = 4$), then $\text{rev}_k(3) = 12$, since the 4-bit representation of 3 is 0011, which when reversed gives 1100, the 4-bit representation of 12. Therefore, to get the bit-reversal permutation of $B[0..n-1]$, we need to swap $B[3]$ with $B[12]$.

(a) (4 points) Given a function rev_k that runs in $O(k)$ time, write an algorithm to perform the bit-reversal permutation on an array of length $n = 2^k$ in $O(nk)$ time.

We can use an algorithm based on an amortized analysis to improve the running time of the bit-reversal permutation. We maintain a "bit-reversed counter" and a procedure BITREVERSEDINCREMENT that, when given a bit-reversed-counter value a, produces $\text{rev}_k(\text{rev}_k(a) + 1)$. If $k = 4$, for example, and the bit-reversed counter starts at 0, then successive calls to BITREVERSEDINCREMENT produce the sequence

0000, 1000, 0100, 1100, 0010, 1010, … = 0, 8, 4, 12, 2, 10, ….

This means, to get the bit-reversal permutation of $B[0..n-1]$, we need to swap $B[0]$ with $B[0]$, $B[1]$ with $B[8]$, $B[2]$ with $B[4]$, $B[3]$ with $B[12]$, $B[5]$ with $B[10]$, ….

That is, BITREVERSEDINCREMENT(a) = $\text{rev}_k(\text{rev}_k(a) + 1)$. At the beginning the counter $a = 0$ (so we swap B[0] with B[0]). Then the 1st call gets the counter a to be BITREVERSEDINCREMENT(0) = $\text{rev}_k(\text{rev}_k(0) + 1) = 8$ (so we swap B[1] with B[8]). The 2nd call gets the counter a to be BITREVERSEDINCREMENT(8) = $\text{rev}_k(\text{rev}_k(8) + 1) = 4$ (so we swap B[2] with B[4]). The 3rd call gets the counter a to be BITREVERSEDINCREMENT(4) = $\text{rev}_k(\text{rev}_k(4) + 1) = 12$ (so we swap B[3] with B[12]). ...

(b) (4 points) Suppose we store the bit-reversed counter in an array $A[0..k-1]$ of bits, that is,

$a = \sum_{i=0,\ldots,k-1} A[i] 2^i$.

Describe an implementation of the BITREVERSEDINCREMENT procedure on A that allows the bit-reversal permutation on an n-element array to be performed in time $O(n)$. You may assume that, given the bit-array A, the element $B[a] = B[A]$ can be accessed directly in $O(1)$ (no need to explicitly compute a).

(c) (7 points) Describe the $O(n)$-time bit-reversal permutation using the BITREVERSEDINCREMENT procedure in (b). Justify the running time.

(a)

The function $\text{rev}_k(a)$ can be implemented by simply putting all the bits of a reversely. Since a has k bits, this is in $O(k)$.

The bit reversal permutation on array $B[0..n-1]$ is implemented as follows: for each index $i = 0, \ldots, n-1$, compute $j = \text{rev}_k(i)$, and swap $B[i]$ with $B[j]$ if $i < j$. In this way, each element is put in the right position. Since $\text{rev}_k(\text{rev}_k(a)) = a$, each pair of elements in reversed indices are swapped just once.

Another way is to guarantee one swap for each pair is to use an auxiliary bitmap array of length n, where the i-th element in it is a bit indicating whether $B[i]$ is already swapped.
Or we can use an auxiliary array B' to store the elements of B into the new position.

(b)
The implementation of BITREVERSEDINCREMENT is similar to INCREMENT; the only change is in the loop to go through the bits reversely.

```
BITREVERSEDINCREMENT(A)
    i = A.length − 1
    while i ≥ 0 and A[i] == 1
        A[i] = 0
        i = i − 1
    if i ≥ 0
        A[i] = 1
```

(c)
The bit reversal permutation on array $B[0..n-1]$ is implemented the same way as in (a), except that when computing $j = \text{rev}_k(i)$, we use BITREVERSEDINCREMENT(A) defined in (b).

Suppose $k = A.length$. During the $n = 2^k$ calls of BITREVERSEDINCREMENT, the leftmost bit $A[k-1]$ is flipped n times. Similarly, $A[k-2]$ is flipped $n/2$ times, and so on. In general, the bit $A[k-i-1]$ is flipped $n / 2^i$ times, for $i = 0, \ldots, k-1$. Thus, the total number of flips in the n calls is

$$n + n/2 + \ldots + 1 = 2n.$$

Therefore, the total time of n BITREVERSEDINCREMENT is $O(n)$. Since the swapping of elements in B takes $O(n)$, the time for bit reversal permutation is $O(n)$.

5. (*15 points*) Making binary search dynamic

Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. We can improve the time for insertion by keeping several sorted arrays.

Specifically, suppose that we wish to support SEARCH and INSERT on a set of n elements. Let

$$k = \lceil \log(n+1) \rceil,$$

and let the binary representation of n be $(n_{k-1}, n_{k-2}, \ldots, n_0)$. We have k sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where for $i = 0, 1, \ldots, k-1$, the length of array A_i is 2^i. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefore

$$\sum_{i=0,\ldots,k-1} n_i 2^i = n.$$

Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

(a) (4 points) Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.
(b) (7 points) Describe how to perform the INSERT operation. Analyze its worst-case and amortized running times.
(c) (4 points) Describe how to perform the DELETE operation for this data structure. Analyze its worst-case running time.

(a)
For searching, we do binary search on each array one by one; return the target as long as it is found, and return FALSE otherwise. In the worst case, when the target does not exist, we need to go through all arrays. The worst-case running time is

$$O(\log(n) + \log(n/2) + \ldots + 2 + 1)$$
$$= O(\log(n) + (\log(n) - 1) + \ldots + 2 + 1)$$
$$= O(\log^2(n))$$

(b)
Suppose we are inserting the $(i + 1)$-th element, and consider the binary representation of i and $i + 1$. Suppose i has k 1's on its rightmost side and its k+1 rightmost bit is 0. Increment i to $i + 1$ would flip its rightmost k bits from 1 to 0, and its k+1 rightmost bit from 0 to 1. Correspondingly, we will merge the first k arrays (of size $2^0, 2^1, \ldots, 2^{k-1}$) together with the element being inserted and generate an array of size 2^k. This merging takes time $O(2^k)$.

In general, for n insertions, the array of size $1 = 2^0$ is merged at most $n/2 + 1$ times, and the array of size 2^{k-1} is merged $n/2^k + 1$ times. So the total running time for n insertions is (let $k = \lceil \log(n + 1) \rceil$)

$$O(n/2 + 2 * n/4 + \ldots + 2^{k-1} * n/2^k)$$
$$= O(n * k)$$
$$= O(n\log(n))$$

Thus, the amortized cost of each insertion is $O(\log(n))$.

(c)
To delete an element, suppose we find it in array A_i, and assume the smallest non-empty array is A_j. We will do the following:

- Remove the target from A_i, and also insert into A_i an arbitrary element a of A_j. (The size of A_i will not change.)
- Remove a from A_j, and break the rest of A_j into smaller arrays of sizes $1, 2, \ldots, 2^{j-1}$.

The worst-case cost is $O(n)$; since the largest array may have size n, breaking it into smaller arrays takes linear time, and inserting an element into it respecting the order also takes linear time.

6. (*10 points*) BFS on graphs

(a) (5 points) Consider an *undirected* graph with 7 vertices: {1, 2, …, 7}, and the following 10 *undirected* edges:

{ (1, 2), (1, 3), (1, 4), (2, 3), (2, 4),
(2, 5), (3, 4), (4, 5), (5, 6), (5, 7) }.

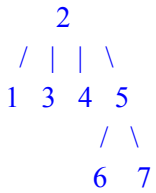
Draw the breath-first tree after running breadth-first search on this graph with source vertex 2.

(b) (5 points) Consider a *directed* graph with 7 vertices: {1, 2, …, 7}, and the following 10 *directed* edges:

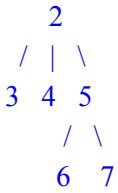
(1, 2), (1, 3), (1, 4), (2, 3), (2, 4),
(2, 5), (3, 4), (4, 5), (5, 6), (5, 7).

Draw the breath-first tree after running breadth-first search on this graph with source vertex 2.

(a)



(b)



7. (*10 points*) Shortest-paths in undirected graphs

We consider undirected graphs (with no weights). Often there are multiple shortest paths between two nodes of a graph. For example, in a graph with vertices {1, 2, 3, 4, 5} and edges

{(1, 2), (1, 3), (2, 4), (3, 4), (4, 5)},

there are two shortest paths from 1 to 5:

(1, 2, 4, 5), and
(1, 3, 4, 5).

Describe a linear-time algorithm such that, given an undirected, unweighted graph and two vertices u and v, the algorithm counts the number of distinct shortest paths from u to v. Justify its correctness and the running time.

The following is the pseudocode of BFSCount extending BFS; it uses an attribute *count* to store the number of shortest paths from the source to each vertex.

BFSCount(G, s)

```
for each u ∈ G.V
        u.color = WHITE
        u.d = ∞
        u.π = NIL
        u.count = 0
s.color = GRAY
s.d = 0
s.count = 1
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ G.Adj[u]
                if v.color == WHITE
                        v.color = GRAY
                        v.d = u.d + 1
                        v.π = u
                        v.count = u.count
                        ENQUEUE(Q, v)
                elseif v.d == u.d + 1
                        v.count = v.count + u.count
        u.color = BLACK
```

To get the number of shortest paths from u to v, we can run BFSCount(G, u) first, and return $v.count$. The running time is linear $O(|V| + |E|)$, the same as BFS. (The outer loop runs $|V|$ times, and the inner loop runs in total $2|E|$ times.)

The correctness is analyzed in the following. For convenience, we define the number of shortest paths from s to itself to be 1. We first show is simple lemma.

Lemma. **The number of paths from s to v of length d is the sum of the number of paths of length $d - 1$ from s to each neighbor of v.**
Proof. Firstly, each path from s to v of length d must have some neighbor of v appearing immediately v, and the length of the subpath from s to this preceding neighbor is $d - 1$.
Secondly, any two paths from s to v of length d with the different preceding vertex before v are different paths. (End of proof.)

Theorem. **For each v, BFSCount computes the number of shortest paths from s to v correctly; that is, BFSCount computes $v.count$ as the number of paths from s of length $v.d$.**
Proof. The proof is by induction on dequeuing operations.

For the base case, when the source s is dequeued, its *count* is 1, unchanged from its initialized value.

Consider a vertex v at the moment when it is dequeued; the induction hypothesis is that all vertices dequeued before v have their *count* value correctly computed. Since $v.count$ is updated only when an edge (w, v) is examined, we consider all such edges leading to v.

Consider when we are exploring the edge (w, v):

- If w is dequeued before v, then by the induction hypothesis, $w.count$ is the number of paths from s to w of length $w.d$.
    - If v is WHITE, then by the correctness of BFS, we will set $v.d = w.d + 1$ and this is the distance from s to v. So we initialize $v.count = w.count$.
    - If v is GRAY, then by the correctness of BFS, $v.d$ is already set to be the distance from s to v; if $w.d + 1 == v.d$, then any path from s to w of length $w.d$ followed by (w, v) gives a path from s to v of length $w.d + 1 = v.d$. So we update $v.count = v.count + w.count$.
    - v cannot be BLACK, since it is dequeued (enqueued) after w is already dequeued.
- If w is dequeued after v, we next argue that $v.count$ will not be updated anymore. (Note that in this case, the induction hypothesis no longer holds.) By the correctness of BFS, when v was enqueued, $v.d$ was already correctly set as the distance from s to v. For any vertex w enqueued (dequeued) after v, we have $w.d \geq v.d$ (by the Corollary in Lecture Slides#19). Therefore, $w.d + 1 > v.d$, so $w.count$ will not be updated when visiting any such w.

Therefore, we compute $v.count$ as the sum of the number of paths of length $v.d - 1$ from s to each incoming neighbor of v, which is the number of shortest paths (paths of length $v.d$) from s to v.
(End of proof.)